TEMA 4.

ÁRBOLES

4.1. CONCEPTOS GENERALES.

Un árbol es una estructura de datos ramificada (no lineal) que puede representarse como un conjunto de **nodos** enlazados entre sí por medio de **ramas**. La información contenida en un nodo puede ser de cualquier tipo simple o estructura de datos.

Los árboles permiten modelar diversas entidades del mundo real tales como, por ejemplo, el índice de un libro, la clasificación del reino animal, el *árbol* genealógico de un apellido, etc.

La figura 4.1. muestra un ejemplo de estructura en árbol (la numeración de los nodos es arbitraria). Se entiende por "topología" de un árbol a su representación geométrica.

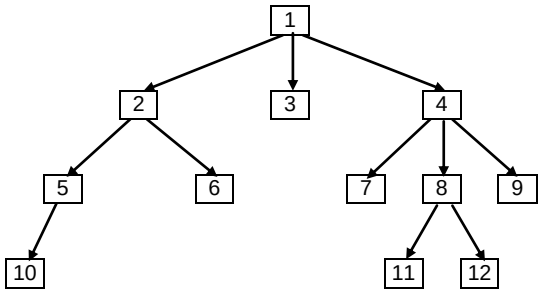

Figura 4.1. Ejemplo de árbol.

Una definición formal es la siguiente:

Un árbol es una estructura de datos base que cumple una de estas dos condiciones:

- *Es una estructura vacía, o*
- *Es un nodo de tipo base que tiene de 0 a N subárboles disjuntos entre sí.*

Al nodo base, que debe ser único, se le denomina **raíz** y se establece el convenio de representarlo gráficamente en la parte superior.

En un árbol se representa una relación jerárquica a partir del nodo raíz en sentido vertical descendente, definiendo **niveles**[1]. El nivel del nodo raíz es 1.

Desde la raíz se puede llegar a cualquier nodo progresando por las **ramas** y atravesando los sucesivos niveles estableciendo así un **camino**. En la figura 4.1. el nodo 7 está a nivel 3 y la secuencia de nodos 4, 8 y 11 constituye un (sub)camino.

Se dice que un nodo es **antecesor** de otro cuando ambos forman parte de un camino y el primero se encuentra en un nivel superior (numeración más baja) al del segundo (numeración más alta). En el ejemplo anterior el nodo 4 es antecesor del 11. Por el contrario, el nodo 11 es **descendiente** del 4.

La relación entre dos nodos separados de forma inmediata por una rama se denomina **padre/hijo**. En el ejemplo de la figura 4.1., el nodo 5 es hijo del nodo 2 y, recíprocamente, el nodo 2 es padre del nodo 5. En un árbol un padre puede tener varios hijos pero un hijo solo puede tener un padre.

Se denomina **grado** al número de hijos de un nodo. Por ejemplo, en la figura 4.1. el nodo 4 tiene grado 3 y el nodo 7 tiene grado 0.

Se dice que un nodo es **hoja** cuando no tiene descendientes (grado 0).

Se establecen los siguientes atributos para un árbol:

- Altura / profundidad / nivel: La mayor altura / profundidad / nivel de sus nodos. La altura del árbol de la figura 4.1. es 4 (la alcanzan sus nodos 10, 11 y 12).
- Amplitud / Anchura: El número de nodos del nivel más poblado. En el ejemplo, 5 (nivel 3).
- Grado: el mayor de los grados de los nodos. En el ejemplo, 3 (nodos 1 y 4).

[1] Los términos “altura” o “profundidad” son sinónimos del de nivel.

Finalmente, indicar que se dice que un árbol es **completo** cuando todos sus nodos (excepto las hojas) tienen el mismo grado y los diferentes niveles están poblados por completo. A veces resulta necesario completar un árbol añadiéndole nodos **especiales**. La figura 4.2. representa el resultado de completar el árbol de la figura 4.1. (en la figura los nodos especiales aparecen sombreados)

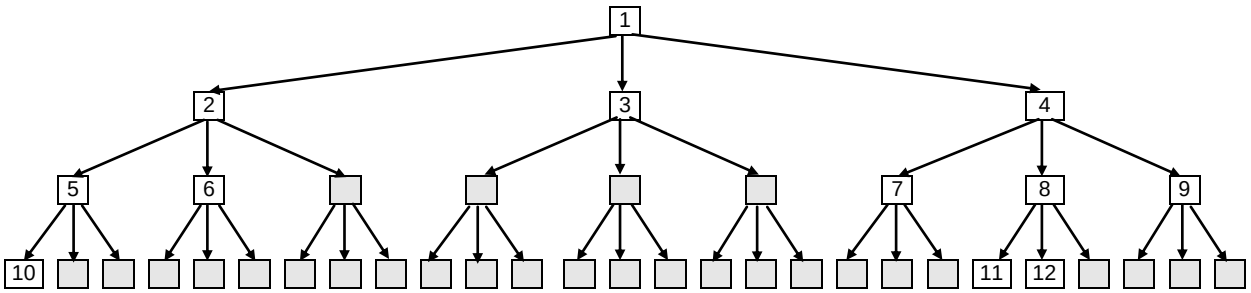

Figura 4.2. Árbol completo de grado 3.

En un árbol completo de grado *g*, la amplitud del nivel *n* es $A_n == g^{n-1}$, y el número total de nodos del árbol es:

$$N_n = \frac{g^n - 1}{g - 1}.$$

4.2. ÁRBOLES BINARIOS.

Se definen como árboles de grado 2. Esto es, cada nodo puede tener dos, uno o ningún hijo. Al tratarse como mucho de dos hijos, cada uno de ellos puede identificarse como **hijo izquierdo** o **hijo derecho**.

4.2.1. Implementación física.

El gráfico de un árbol es una representación conceptual cuya implementación física admite diversas posibilidades condicionadas, en primer lugar, por el dispositivo de almacenamiento del mismo (memoria principal o memoria externa). A los efectos del curso nos ocuparemos exclusivamente de la memoria principal en donde puede optarse por dos filosofías principales:

- Estructuras de datos estáticas, normalmente matrices.
- Estructuras de datos dinámicas

En cualquier caso, la representación física de un árbol requiere contemplar tres componentes:

- La clave (simple o compuesta).
- Dos punteros, indicando las ubicaciones respectivas de los nodos hijo izquierdo e hijo derecho[2].

Ejemplo. La figura 4.3. representa un árbol binario que podría implementarse físicamente según se ilustra en las figuras 4.4. (estructura estática) ó 4.5. (estructura dinámica).

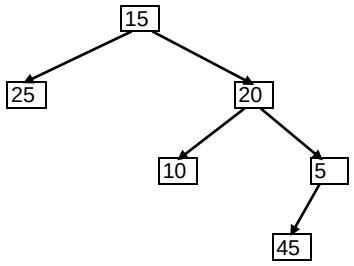

Figura 4.3. Ejemplo de árbol binario.

	0	1	2	3	4	5
Clave	15	20	5	25	45	10
Hijo izquierdo	3	5	4	*	*	*
Hijo derecho	1	2	*	*	*	*

Figura 4.4. Ejemplo de árbol binario. Implementación estática.

[2] Deberá establecerse un convenio para indicar la inexistencia de hijo(s).

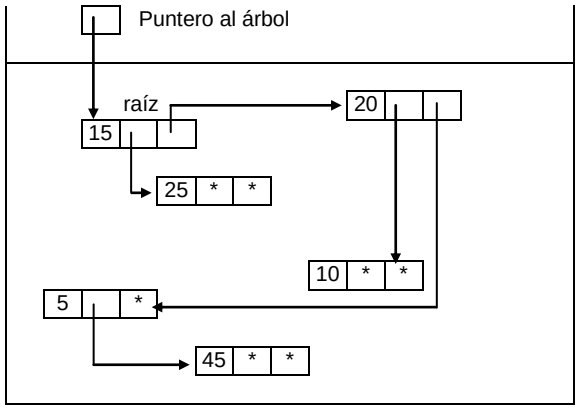

Figura 4.5. Ejemplo de árbol binario. Implementación dinámica.

4.2.2. Algoritmos básicos con árboles binarios[3].

Para la utilización de árboles binarios es necesario definir las clases NodoArbol y Arbol siguiendo la sintaxis siguiente:

```
public class NodoArbol {
   public NodoArbol (int dato) {
      clave = dato;
      iz = null;
      de = null;
   }
   public int clave;
   public NodoArbol iz, de;
}
public class Arbol {
   public NodoArbol raiz;
   public Arbol () {
      raiz = null;
   }
}
```

4.2.2.1.Recorridos.

Se entiende por recorrido el tratamiento realizado para acceder a los diferentes nodos de un árbol. El recorrido puede afectar a la totalidad de los nodos del árbol (recorrido completo), por ejemplo si se desea conocer el número de nodos, o finalizar anticipadamente en función de determinada/s circunstancia/s, por ejemplo al encontrar el valor de una clave determinada.

En cualquier caso, el recorrido se puede realizar basándose en las siguientes modalidades:

[3] Los siguientes algoritmos se han desarrollado en Java considerando la implementación del árbol por medio de una estructura dinámica. A efectos de la realización de prácticas se utiliza la sintaxis del lenguaje Java.

- En profundidad. Se progresa verticalmente desde la raíz hacia las hojas y, en principio, de izquierda a derecha. Se plantean tres posibilidades atendiendo al momento en que se procesa la clave.

    - Preorden. La clave se procesa en primer lugar. Posteriormente se realizan las llamadas recursivas (por la izquierda y por la derecha). La exploración en preorden del árbol de la figura 4.3.a. daría el siguiente resultado: 15, 25, 20, 10, 5, 45.

      Este tipo de recorrido es el más utilizado pues atiende a la estructura de organización jerárquica del árbol. También es el más eficiente cuando se trata de buscar claves.

      En general, el algoritmo de recorrido en preorden es el siguiente:

```
// Escribe las claves del árbol binario de raiz "a" en preorden.
static void preOrden (NodoArbol arbol) {
   if (arbol != null) {
      System.out.print (arbol.clave + " ") ;          // Nodo
      preOrden (arbol.iz);                            // Izquierda
      preOrden (arbol.de);                            // Derecha
   }
}
public void preOrden () {
   preOrden (raiz);
}
```

    - Orden central. Se desciende recursivamente por la rama izquierda. Al alcanzar la condición de finalización (*arbol == null*) se retorna y se procesa la clave, y a continuación se progresa por la rama derecha. La exploración en orden central del árbol de la figura 4.3.a. daría el siguiente resultado: 25, 15, 10, 20, 45, 5.

      Este tipo de recorrido es especialmente útil en todos aquellos procesos en los que sea necesario recorrer los nodos de acuerdo al orden físico de los mismos.

      En general, el algoritmo de recorrido en orden central es el siguiente:

```
// Escribe las claves del árbol binario en orden central.
static void ordenCentral (NodoArbol arbol) {
   if (arbol != null) {
      ordenCentral (arbol.iz);                        // Izquierda
      System.out.print (arbol.clave + " ") ;          // Nodo
      ordenCentral (arbol.de);                        // Derecha
   }
}
public void ordenCentral () {
   ordenCentral (raiz);
}
```

- Postorden. Se desciende recursivamente por la rama izquierda, al alcanzar el final de dicha rama, se retorna y se desciende por la rama derecha. Cuando se alcanza el final de la rama derecha, se procesa la clave. La exploración en postorden del árbol de la figura 4.3.a. daría el siguiente resultado: 25, 10, 45, 5, 20, 15.

  Este recorrido es el menos utilizado. Se utiliza para liberar memoria, o bien cuando la información de los niveles más profundos del árbol afecta a la información buscada.

  En general, el algoritmo de recorrido en postorden es el siguiente:

```
// Escribe las claves del árbol binario en postorden.
static void postOrden (NodoArbol arbol) {
   if (arbol != null) {
      postOrden (arbol.iz);                          // Izquierda
      postOrden (arbol.de);                          // Derecha
      System.out.print (arbol.clave + " ") ;         // Nodo
   }
}
public void postOrden () {
   postOrden (raiz);
}
```

Ejemplo de algoritmo de recorrido en profundidad: desarrollar un método estático (sumaClaves) que obtenga la suma de las claves del árbol[4].

```
static int sumaClaves (NodoArbol arbol) {
   int resul;
   if (arbol != null)
      resul = arbol.clave + sumaClaves (arbol.iz) + sumaClaves (arbol.de);
   else resul = 0;
   return resul;
}
static int sumaClaves (Arbol a) {
   return sumaClaves (a.raiz);
}
```

Como ejercicio se propone al alumno la realización de un algoritmo que cuente los nodos que tiene un árbol, utilizando uno de los tres recorridos en profundidad propuestos.

[4] Puede realizarse indistintamente con cualquiera de las modalidades de recorrido, en la solución propuesta se hace un recorrido en preorden.

- Recorrido en amplitud. Implica un acceso a las claves recorriendo cada nivel de izquierda a derecha y descendiendo al siguiente. Por ejemplo, la exploración en amplitud del árbol de la figura 4.3.a. daría el siguiente resultado: 15, 25, 20, 10, 5, 45.

Para la implementación del recorrido en amplitud se requiere la utilización de la técnica iterativa así como de la disponibilidad de una estructura de datos auxiliar: TAD cola de nodos de árbol:

```
class NodoCola {
   NodoArbol dato;
   NodoCola siguiente;
   // Constructores
   NodoCola (NodoArbol elemento, NodoCola n) {
      dato = elemento;
      siguiente = n;
   }
}
```

Por ejemplo, el algoritmo siguiente permite obtener un listado de las claves de un árbol binario recorrido en amplitud:

```
   static void listarAmplitud (NodoArbol arbol) {
      NodoArbol p;
      Cola c = new TadCola ();
      p = arbol;
      if (p != null)
         c.encolar (p);
      while (! c.colaVacia ()) {
         p = c.desencolar ();
         System.out.print (p.clave + " ");
         if (p.iz != null)
            c.encolar (p.iz);
         if (p.de != null)
            c.encolar (p.de);
      }
   }
   public void listarAmplitud () {
      listarAmplitud (raiz);
   }
```

La idea, básicamente, consiste en encolar en la cola de punteros las direcciones (referencias) de los posibles hijos de cada nodo procesado (su dirección se encuentra en la variable *p*).

A continuación, en cada iteración se extrae el primer elemento de la cola que se corresponde con el nodo actual del árbol. Si su hijo izquierdo y/o su hijo derecho son distintos de null se encolan, y el proceso terminará cuando la cola se encuentre vacía.

4.2.2.2.Búsqueda.

Hasta ahora se han presentado distintos modos de recorrer el árbol completo. En algunas ocasiones, será necesario finalizar antes de completar el recorrido, no realizando posteriores llamadas recursivas. En algunos casos se requiere el empleo de un argumento adicional (pasado por referencia) que permita evitar, en su caso, la ejecución de posteriores llamadas recursivas.

El siguiente ejemplo implementa un método (*búsqueda*) que recibe como argumentos un árbol y una clave (ambos pasados por valor) y devuelve *true* si dicha clave se encuentra en el árbol, y *false* en caso contrario.

Se procede a recorrer en principio todo el árbol. Cada vez que se alcanza un extremo (llamada desde una hoja) se cumple la condición *arbol == null* y el método devuelve el valor *false*. Si a lo largo del recorrido se encuentra la clave buscada, el método devolverá el puntero a dicho nodo, y ya no se realizan nuevas llamadas recursivas, siendo este valor el que se entrega al módulo que llamó a el método, lo que evita el error que supondría alcanzar la condición de finalización en el extremo derecho del árbol (hijo derecho de la última hoja).

Por razones de eficiencia se realiza un recorrido en preorden.

```
static boolean busqueda (NodoArbol arbol, int dato) {
   boolean resul = false;
   if (arbol != null)
      if (arbol.clave == dato)
         resul = true;
      else {
         resul = busqueda (arbol.iz, dato);
         if (!resul)
             resul = busqueda (arbol.de, dato);
      }
   return resul;
}
public boolean busqueda (int dato) {
   return busqueda (raiz, dato);
}
```

4.2.2.3. Creación de un árbol.

Para insertar claves en un árbol es necesario establecer previamente un criterio. Normalmente no se permite insertar claves repetidas. En el apartado 4.3.2.1. se muestra cómo insertar nodos en un árbol binario de búsqueda. Para los árboles binarios que no sean de búsqueda, se puede utilizar el método juntar, que recibe una clave y dos árboles (*a1* y *a2*), y devuelve un árbol con la clave en la raíz, *a1* como subárbol izquierdo, y *a2* como subárbol derecho:

```
   public void juntar (int dato, Arbol a1, Arbol a2) {
      if (a1.raiz == a2.raiz && a1.raiz != null) {
         System.out.println ("no se pueden mezclar, t1 y t2 son iguales") ;
         return;
      }
      // Crear el nodo nuevo
      raiz = new NodoArbol (dato, a1.raiz, a2.raiz) ;
      // Borrar los árboles a1 y a2
      if (this != a1)
         a1.raiz = null;
      if (this != a2)
         a2.raiz = null;
   }
```

En el siguiente ejemplo, se muestra como generar el árbol de la figura 4.6:

```
   public static void main (String [] args) {
      Arbol a1 = new Arbol (1) ;
      Arbol a3 = new Arbol (3) ;
      Arbol a5 = new Arbol (5) ;
      Arbol a7 = new Arbol (7) ;
      Arbol a2 = new Arbol ();
      Arbol a4 = new Arbol ();
      Arbol a6 = new Arbol ();
      a2.juntar (2, a1, a3) ;
      a6.juntar (6, a5, a7) ;
      a4.juntar (4, a2, a6) ;
}
```

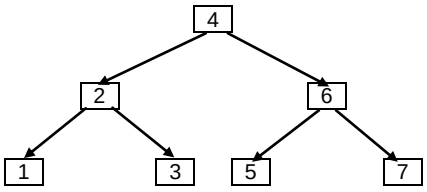

Figura 4.6. Ejemplo de árbol binario.

4.2.2.4.Tratamiento de hojas.

En este tipo de algoritmos se trata de evaluar la condición de que un nodo del árbol sea una hoja:

```
(arbol.iz == null) && (arbol.de == null).
```

Por ejemplo, el siguiente método de tipo entero (*cuentaHojas*) devuelve como resultado el número de hojas del árbol que se le entrega como argumento.

```
static int cuentaHojas (NodoArbol arbol) {
   int resul = 0;
   if (arbol != null)
      if ((arbol.iz == null) && (arbol.de == null))
         resul = 1;
      else resul = cuentaHojas (arbol.iz) + cuentaHojas (arbol.de);
   return resul;
}
static int cuentaHojas (Arbol a) {
   return cuentaHojas (a.raiz);
}
```

4.2.2.5. Procesamiento con constancia del nivel.

Determinados tratamientos requieren conocer el nivel en que se encuentran los nodos visitados, para lo cual es necesario conocer el nivel actual.

Recorrido en profundidad.

Cuando se va a realizar un tratamiento en profundidad, pasamos como argumento (por valor) el nivel actual. En cada llamada recursiva (por la derecha o por la izquierda) se incrementa el nivel en una unidad. La inicialización de dicho argumento (a 1) se realiza en la primera llamada (no recursiva).

Por ejemplo, el siguiente método (*clavesNiveles*) recorre un árbol en preorden[5] mostrando en la pantalla el valor de las diferentes claves así como el nivel en que se encuentran.

```
   static void clavesNiveles (NodoArbol arbol, int n) {
      if (arbol != null) {
         System.out.println ("Clave: " + arbol.clave + " en el nivel: " + n);
         clavesNiveles (arbol.iz, n+1);
         clavesNiveles (arbol.de, n+1);
      }
   }
   static void clavesNiveles (Arbol a) {
      clavesNiveles (a.raiz, 1);
   }
```

Recorrido en amplitud.

El algoritmo explicado para el recorrido en amplitud en el apartado 3.2.2.2., permite recorrer el árbol en amplitud, sin embargo, no se tiene constancia del nivel en que se encuentran los nodos a los que se accede.

En caso de que esto sea necesario debería modificarse, bien la estructura de la cola añadiendo el nivel, o bien el algoritmo visto anteriormente de manera que su estructura sea una iteración anidada en dos niveles.

[5] La modalidad de recorrido es indiferente.

Si se puede cambiar la estructura de la cola, podría utilizarse una cola con el siguiente tipo de elementos:

```
TElemento (NodoArbol a, int n) {
   nivel = n;
   arbol = a;
}
NodoColaN (NodoArbol a, int n, NodoColaN sig) {
   elem = new TElemento (a, n);
   siguiente = sig;
}
```

Utilizando los siguientes métodos para obtener el árbol y el nivel:

```
public NodoArbol obtenerArbol (TElemento e) {
   return e.arbol;
}
public int obtenerNivel (TElemento e) {
   return e.nivel;
}
```

De tal forma que al encolar el árbol completo, pondríamos el nivel a 1, y posteriormente, cada vez que se va a encolar un elemento, lo primero que haremos será actualizar el nivel.

```
static void listarAmplitudNiveles (NodoArbol arbol) {
   NodoArbol p;
   int n;
   ColaN c = new TadColaN ();
   p = arbol;
   if (p != null) {
      TElemento elem = new TElemento (p, 1);
      c.encolar (elem);
      while (! c.colaVacia ()) {
         elem = c.desencolar ();
         p = elem.obtenerArbol ();
         n = elem.obtenerNivel ();
         System.out.println ("nivel: " + n + " " + p.clave + " ");
         if (p.iz != null) {
            elem = new TElemento (p.iz, n+1);
            c.encolar (elem);
         }
         if (p.de != null){
            elem = new TElemento (p.de, n+1);
            c.encolar (elem);
         }
      }
   }
}
public void listarAmplitudNiveles (){
   listarAmplitudNiveles (raiz);
}
```

Si no se puede alterar la estructura de la cola, sería necesario cambiar el algoritmo de manera que existan dos niveles de iteración. El nivel externo es el general (para todo el árbol) y el nivel interno para cada nivel del árbol. Cuando se inicia el recorrido de un nivel

(controlado por medio de la variable *contador*) es necesario conocer a priori su amplitud (variable *actual*) y según se va explorando dicho nivel se cuenta el número de hijos del siguiente nivel (variable *siguiente*). Al iniciar el proceso del siguiente nivel se pasa a la variable *actual* el último valor encontrado en la variable *siguiente*.

A continuación se muestra como ejemplo una variante del ejemplo anterior.

```
static void listarAmplitudNiveles (NodoArbol arbol) {
   NodoArbol p;
   Cola c = new TadCola ();
   int actual, siguiente, contador, altura;
   altura = 0;
   siguiente = 1;
   p = arbol;
   if (p != null )
       c.encolar (p);
   while (!c.colaVacia()) {
      actual = siguiente;
      siguiente = 0;
      contador = 1;
      altura++;
      while (contador <= actual) {
         p = c.desencolar ();
         System.out.println ("clave: " + p.clave + " nivel: " + altura);
         contador++;
         if (p.iz != null) {
            c.encolar (p.iz);
            siguiente++;
         }
         if (p.de != null) {
            c.encolar (p.de);
            siguiente++;
         }
      }
   }
}
public void listarAmplitudNiveles (){
   listarAmplitudNiveles (raiz);
}
```

4.2.3. Ejemplo.

A continuación se desarrolla un método que intenta verificar si dos árboles son iguales o no.

```
static boolean iguales (NodoArbol a, NodoArbol b) {
   boolean resul ;

   if ((a == null) && (b == null))
      resul = true;
   else if ((a == null) || (b == null))
         resul = false;
      else if (a.clave == b.clave)
            resul = iguales(a.iz, b.iz) && iguales (a.de, b.de);
         else resul = false;
   return resul;
}
static boolean iguales (Arbol a1, Arbol a2) {
   return iguales (a1.raiz, a2.raiz);
}
```

4.3. ÁRBOLES BINARIOS DE BÚSQUEDA.

Un árbol binario de búsqueda es un tipo de árbol binario en el que se verifica para todo nodo que las claves de su subárbol izquierdo son menores que las de dicho nodo y las claves de su subárbol derecho son mayores. La figura 4.10. muestra un ejemplo.

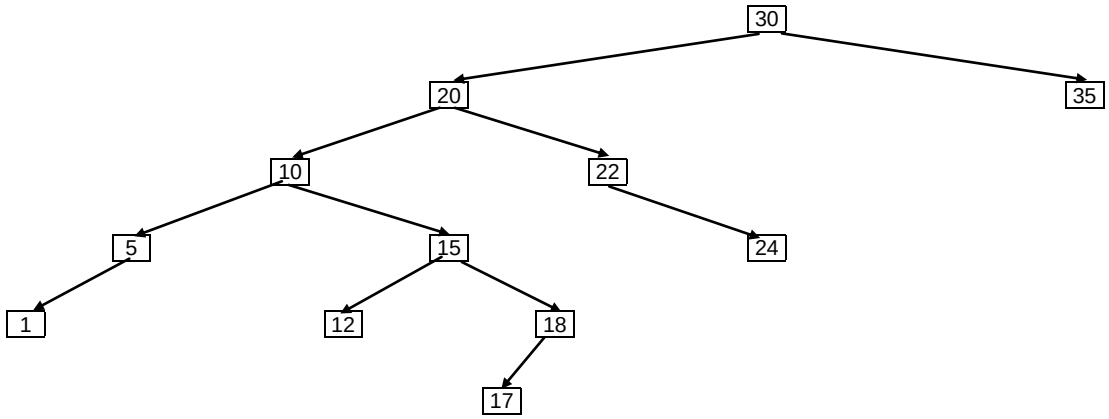

Figura 4.7. Ejemplo de árbol binario de búsqueda.

La utilización de árboles binarios de búsqueda implica en términos generales las siguientes consideraciones:

4.3.1. Algoritmos de consulta.

En caso de búsqueda de alguna clave el proceso resulta de mayor eficiencia que en el caso general, dado que el encaminamiento se produce de forma dirigida (por la derecha si *arbol.clave* < *dato* y por la izquierda si *arbol.clave* > *dato*). La condición de finalización pesimista es alcanzar un extremo del árbol (*arbol == null*) que en caso de alcanzarse (una sola vez, no varias como podría suceder para árboles binarios genéricos) implicaría que no se ha encontrado la clave buscada y el proceso finalizaría.

La terminación anticipada (no es necesario hacer más llamadas recursivas) se produce cuando se encuentra la clave buscada (*arbol.clave == dato*). A continuación se muestra como ejemplo el método *busqueda* que devuelve un puntero al nodo que contiene la clave coincidente con el argumento *dato* y null en caso de que no exista dicha clave.

```
static boolean busqueda (NodoArbol arbol, int dato) {
   boolean resul = false;
   if (arbol != null)
      if (arbol.clave == dato)
         resul = true;
      else if (arbol.clave > dato)
            resul = busqueda (arbol.iz, dato);
         else resul = busqueda (arbol.de, dato);
   return resul;
}
public boolean busqueda (int dato) {
   return busqueda (raiz, dato);
}
```

En caso de no buscar una clave determinada el recorrido del árbol binario de búsqueda no ofrece ninguna particularidad respecto al árbol binario genérico, pudiendo realizarse dicho recorrido en cualquier modalidad: orden central, preorden o postorden, así como en amplitud.

No obstante, cuando se desea recuperar el conjunto de claves del árbol ordenadas, el recorrido deberá ser en orden central (de izquierda a derecha para secuencia ascendente y de derecha a izquierda para secuencia descendente).

A continuación se muestra como ejemplo el método booleano *esBusqueda* que devuelve *true* si el *arbol* binario, que se pasa como argumento, es de búsqueda y false en caso contrario. Consiste en verificar que la secuencia de claves es ascendente, por lo que habrá que hacer un recorrido en orden central (de izquierda a derecha) y comparar el valor de cada clave con la anterior (*ant*) que, debidamente inicializada, deberá pasarse como argumento en las sucesivas llamadas recursivas. Es necesario identificar el primer elemento de la lista (no tiene elemento anterior con el que comparar la clave), con una variable auxiliar booleana (*primero*), inicializada a true desde fuera de el método.

```
static class Busqueda {
   static int ant;
   static boolean primero = true;
   static boolean esBusqueda (NodoArbol arbol) {
      boolean resul;
      if (arbol == null)
         resul = true;
         else {
            resul = esBusqueda (arbol.iz);
            if (primero)
                primero = false;
            else if (arbol.clave <= ant)
                  resul = false;
            if (resul) {
               ant = arbol.clave;
               resul = esBusqueda(arbol.de);
            }
         }
      return resul;
   }
}
static boolean esBusqueda (Arbol a) {
      return Busqueda.esBusqueda(a.raiz);
   }
```

4.3.2. Algoritmos de modificación.

4.3.2.1. Inserción.

Se trata de crear un nuevo nodo en la posición que le corresponda según el criterio de árbol binario de búsqueda. A continuación se muestra el algoritmo.

```
static NodoArbol insertar (NodoArbol arbol, int dato) {
   NodoArbol resul = arbol;
   if (arbol != null)
      if (arbol.clave < dato)
         arbol.de = insertar (arbol.de, dato);
      else if (arbol.clave > dato)
            arbol.iz = insertar (arbol.iz, dato);
         else System.out.println ("la clave ya existe");
   else resul = new NodoArbol (dato);
   return resul;
}
public void insertar (int dato) {
   raiz = insertar (raiz, dato);
}
```

4.3.2.2. Eliminación.

La eliminación de un nodo en un árbol binario de búsqueda implica una reorganización posterior del mismo con el objeto de que una vez eliminado el nodo el árbol mantenga su naturaleza de búsqueda. Para ello se procede de la manera siguiente:

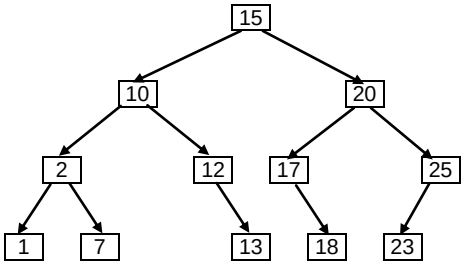

Figura 4.8. Árbol binario de búsqueda

Primero localizamos el nodo a eliminar. Una vez encontrado, tenemos tres posibles casos:

1. Tenemos que borrar un nodo hoja: procedemos a borrarlo directamente. Por ejemplo, si en el árbol de la figura 4.11, queremos borrar la clave 13, el resultado sería la figura 4.12.

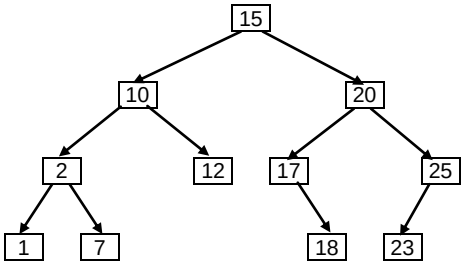

Figura 4.9. Borrado de un nodo hoja en un árbol binario de búsqueda

2. Hay que borrar un nodo con un único descendiente: ascendemos a su descendiente. Por ejemplo, si en el árbol de la figura 4.12, queremos borrar la clave 17, el resultado sería la figura 4.13.

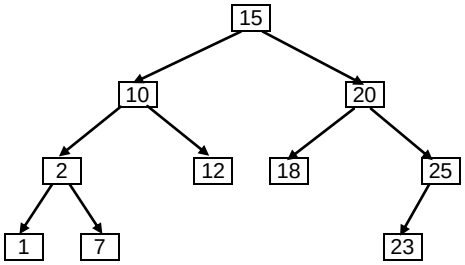

Figura 4.10. Borrado de un nodo con un solo hijo en un árbol binario de búsqueda

3. El nodo que queremos borrar tiene dos hijos. En este caso, sustituimos la clave a borrar por la clave del mayor de sus descendientes menores (el nodo más a la derecha del subárbol izquierdo), y borramos el nodo que tenía anteriormente dicha clave. Para poder hacer esto:

   - Nos situamos en la raíz del subárbol izquierdo de dicho nodo.
   - Se desciende por la derecha de dicho subárbol hasta encontrar un nodo (*n*) sin descendiente derecho.
   - Se sustituye la clave a eliminar por la del nodo *n*.
   - Se elimina el nodo n[6]. Por ejemplo, si se desea eliminar la clave 10 del árbol representado en la figura 4.13., sustituiremos dicha clave por la mayor de sus descendientes menores (el 7), y borraremos el nodo con la clave 7 (el nodo que deberemos borrar ahora será en este caso un nodo hoja, aunque también podríamos encontrarnos con un nodo con un solo descendiente). Como resultado, obtendremos el árbol que se muestra en la figura 4.14.

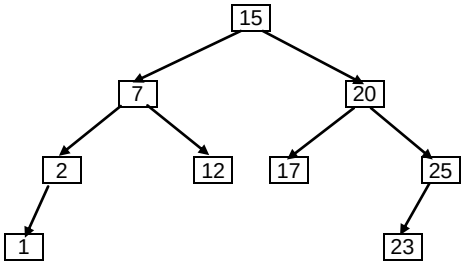

Figura 4.11. Borrado de un nodo con dos hijos en un árbol binario de búsqueda.

A continuación se muestra el código del algoritmo de eliminación.

[6] Obsérvese que se elimina la clave deseada pero no el nodo físico que la albergaba sino otro.

```
static class Eliminar {
   static NodoArbol eliminar2Hijos (NodoArbol arbol, NodoArbol p) {
      NodoArbol resul;
      if (arbol.de != null) {
         resul = arbol;
         arbol.de = eliminar2Hijos (arbol.de, p);
      }
         else {
            p.clave = arbol.clave;
            resul = arbol.iz;
         }
      return resul;
   }
   static NodoArbol eliminarElemento (NodoArbol arbol, int elem) {
   NodoArbol p;
      if (arbol != null)
         if (arbol.clave > elem)
            arbol.iz = eliminarElemento (arbol.iz, elem);
            else if (arbol.clave < elem)
                  arbol.de = eliminarElemento (arbol.de, elem);
               else {
                  p = arbol;
                  if (arbol.iz == null)
                     arbol= arbol.de;
                  else if (arbol.de == null)
                           arbol = arbol.iz;
                        else arbol.iz = eliminar2Hijos (arbol.iz, p);
      }
      else System.out.println ("la clave buscada no existe");
      return arbol;
   }
}
public void eliminar (int elem) {
   raiz = Eliminar.eliminarElemento (raiz, elem);
}
```

4.3.3. Ejemplo.

Dado un árbol binario de búsqueda y una lista enlazada por medio de punteros, implementar un algoritmo en Java que, recibiendo al menos un árbol y una lista ordenada ascendentemente, determine si todos los elementos del árbol están en la lista.

Implementaremos un método booleano con el *arbol* y la *lista* como argumentos. Como la lista está ordenada ascendentemente, y el árbol binario es de búsqueda, realizaremos un recorrido del árbol en orden central, parando en el momento en que detectemos algún elemento que existe en el árbol, pero no en la lista.

```
static boolean estaContenido (NodoArbol arbol, NodoLista lista) {
   boolean seguir, saltar;
   if (arbol == null)
      seguir = true;
   else {
      seguir = estaContenido (arbol.iz, lista);
      if (seguir && (lista != null))
         if (arbol.clave < lista.clave)
            seguir = false;
         else {
            saltar = true;
            while ((lista != null) && saltar)
            if (arbol.clave == lista.clave)
               saltar = false;
            else lista = lista.sig;
            if (!saltar)
               seguir = estaContenido (arbol.de, lista.sig);
            else seguir = false;
         }
      else seguir = false;
   }
   return seguir;
}
static boolean estaContenido (Arbol a, Lista l) {
   return estaContenido (a.raiz, l.inicio);
}
```

4.4. ÁRBOL SOBRE MATRIZ.

4.4.1. Clases y constructores.

Si se desea implementar un árbol binario de búsqueda utilizando una matriz, se pueden definir las siguientes clases NodoArbol y Arbol:

```
class NodoArbol {
   int clave, izq, der;
   NodoArbol () {
      clave = 0;
      izq = -1;
      der = -1;
   }
}
public class ArbolMatriz {
   final int NULL = -1, N = 10;
   NodoArbol [ ] matriz;
   int numNodos;
   ArbolMatriz () {
      matriz = new NodoArbol [N];
      for (int i = 0; i < N; i++)
         matriz [i] = new NodoArbol ();
      numNodos = 0;
   }
...
}
```

Utilizando las clases anteriores, utilizaríamos la siguiente matriz para guardar el contenido del árbol de la figura 4.12.:

	0	1	2	3	4	5	6	7	8	9
Clave	10	20	5	15	45	15	0	0	0	0
Hijo izquierdo	3	6	-1	-1	5	-1	-1	-1	-1	-1
Hijo derecho	2	4	-1	-1	-1	-1	-1	-1	-1	-1

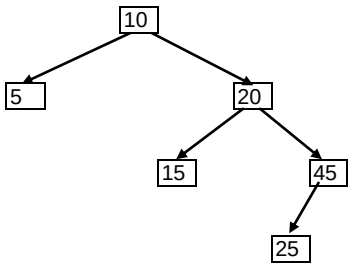

Figura 4.12. Ejemplo de árbol binario.

La clave de la raíz del árbol (10) aparece en la posición 0 de la matriz, en el campo clave. Su hijo izquierdo (el 5), está en la posición 3 de la matriz (dado que matriz [0].izq =3), mientras que el hijo derecho (15) estará en la posición 2. Cuando un elemento es una hoja (por ejemplo, el 15), su hijo izquierdo y su hijo derecho tendrán como valor *NULL* (es decir, -1).

4.4.1.1. Recorridos en profundidad.

A continuación se detallan los códigos correspondientes a los tres recorridos en profundidad. En cualquiera de los casos, desde el método de llamada se llama a uno auxiliar recursivo, pasándole como argumento la posición de la raíz (0).

Recorrido en preorden:

En el método recursivo privado *preOrdenAux*, se escriben las claves del árbol en preorden: primero escribimos la clave del nodo (*matriz [i].clave*), luego se avanza por la izquierda (haciendo la llamada con *matriz [i].izq*), y luego por la derecha (*matriz [i].der*).

```
private void preOrden (int i) {
   if (i != NULL) {
      System.out.print (matriz [i].clave+", ");
      preOrden (matriz [i].izq);
      preOrden (matriz [i].der);
   }
}
public void preOrden () {
   preOrden (0);
}
```

Recorrido en orden central:

```
private void ordenCentral (int i) {
   if (i != NULL) {
      ordenCentral (matriz [i].izq);
      System.out.print (matriz [i].clave+", ");
      ordenCentral (matriz [i].der);
   }
}
public void ordenCentral () {
   ordenCentral (0);
}
```

Recorrido en post-orden:

En el método recursivo privado *ordenCentralAux*, se escriben las claves del árbol en post orden: primero se avanza por la izquierda (haciendo la llamada con *matriz [i].izq,*), luego por la derecha (*matriz [i].der*) y por último escribimos la clave del nodo (*matriz [i].clave*).

```
private void postOrden (int i) {
   if (i != NULL) {
      postOrden (matriz [i].izq);
      postOrden (matriz [i].der);
      System.out.print (matriz [i].clave+", ");
   }
}
public void postOrden () {
   postOrden (0);
}
```

4.4.1.2. Búsqueda.

Para realizar una búsqueda en un árbol binario de búsqueda sobre matriz, es necesario comprobar primero si no hemos llegado a un nodo vacío *(if (i != NULL)*, en cuyo caso devolveremos *false* como resultado*)*, y luego vamos avanzando por la rama correspondiente:

- si la clave es mayor que la buscada seguiremos por la izquierda (*resul = busqueda (matriz [i].izq, dato)*),
- si la clave es menor que la buscada seguiremos por la derecha (*resul = busqueda (matriz [i].der, dato)*),
- y si la clave es la que estamos buscando, pararemos devolviendo como resultado *true*.

```
public boolean busqueda (int dato) {
   return busqueda (0, dato);
}
private boolean busqueda (int i, int dato) {
   boolean resul = false;
   if (i != NULL)
      if (matriz [i].clave > dato)
         resul = busqueda (matriz [i].izq, dato);
      else if (matriz [i].clave < dato)
            resul = busqueda (matriz [i].der, dato);
         else resul = true;
   return resul;
}
```

4.4.1.3. Inserción.

Para realizar la inserción en un árbol binario de búsqueda sobre matriz, es necesario comprobar primero si todavía no está lleno el árbol (comprobando la variable miembro *numNodos*). Si se va a insertar en un árbol vacío, la inserción se hará en el método de llamada (*insertar):*

```
public void insertar (int dato) {
   if (numNodos == 0){
      matriz [0].clave =dato;
      numNodos++;
   }
   else if (numNodos < N)
      insertar (0, dato);
   else System.out.println ("árbol lleno");
}
```

Para el resto de los nodos utilizaremos el método recursivo *insertarAux*. Para ello, primero habrá que localizar dónde deberíamos insertar el nodo (realizando una búsqueda guiada por el árbol). Una vez localizado, utilizaremos para el nuevo nodo la primera posición libre en la matriz (la posición *numNodos*).

```
private int insertar (int i, int dato) {
   int j = NULL;
   if (i != NULL)
      if (matriz [i].clave > dato) {
         j = insertar (matriz [i].izq, dato);
         if (j != NULL)
            matriz [i].izq = j;
         j = i;
      }
      else if (matriz [i].clave < dato) {
            j = insertar (matriz [i].der, dato);
            if (j != NULL)
               matriz [i].der = j;
            j = i;
         }
         else System.out.println ("la clave ya existe");
   else {
      j = numNodos;
      matriz [j].clave = dato;
      numNodos++;
   }
   return j;
}
```

4.4.1.4. Eliminación.

Para eliminar una clave, se sigue la misma estrategia que en el apartado 4.3.2.2:

- El método recursivo *eliminar* busca la clave que se desea eliminar.
- Si es una hoja se elimina directamente,
- Si es un nodo con un hijo se sustituye por el hijo,
- Y si es un nodo con dos hijos se llama al método *eliminarDosHijos* para sustituir la clave del nodo por la inmediatamente anterior (bajando una vez a la izquierda y luego todo el rato a la derecha hasta encontrar un nodo sin hijo izquierdo).

```
public void eliminar (int dato) {
   eliminar (0, dato);
}
private int eliminar (int i, int dato) {
   int j = NULL;
   if (i != NULL)
      if (matriz [i].clave > dato) {
         j = eliminar (matriz [i].izq, dato);
         if (j != NULL)
            matriz [i].izq = j;
         j = i;
      }
      else if (matriz [i].clave < dato) {
            j = eliminar (matriz [i].der, dato);
            if (j != NULL)
                matriz [i].der = j;
             j = i;
          }
          else {
              numNodos--;
              if (matriz [i].der == NULL) {
                 j = matriz [i].izq;
                 borrarNodo (i);
              }
               else if (matriz [i].izq == NULL) {
                        j = matriz [i].der;
                        borrarNodo (i);
                    }
                    else {
                        matriz [i].izq = eliminarDosHijos (matriz [i].izq, i);
                        j = i;
                    }
          }
   else System.out.println ("la clave no existe");
   return j;
}
```

```
private int eliminarDosHijos (int i, int j) {
   int resul = i;
   if (matriz [i].der != NULL)
      matriz [i].der = eliminarDosHijos (matriz [i].der, j);
   else {
      matriz [j].clave = matriz [i].clave;
      borrarNodo (i);
      resul = matriz [i].izq;
   }
   return resul;
}
```

Por último, si encontramos el nodo a eliminar, en cualquiera de los casos (tenga cero, uno o dos hijos) se utilizará un método auxiliar (*borrarNodo*) para desplazar los elementos que haya a continuación del nodo eliminado.

```
private void borrarNodo (int i) {
   for (int j = i; j < numNodos; j++) {
      matriz [j].clave = matriz [j+1].clave;
      matriz [j].der = matriz [j+1].der;
      matriz [j].izq = matriz [j+1].izq;
   }
   for (int j = 0; j <= numNodos; j++) {
      if (matriz [j].der >= i)
         matriz [j].der = matriz [j].der -1;
      if (matriz [j].izq >= i)
         matriz [j].izq = matriz [j].izq -1;
   }
}
```

Por ejemplo, si en el árbol de la figura 4.13., representado por la matriz que aparece a continuación, eliminamos el nodo 20, el resultado sería el de la figura 4.14:

	0	1	2	3	4	5	6	7	8	9
Clave	10	20	5	15	45	25	0	0	0	0
Hijo izquierdo	2	3	-1	-1	5	-1	-1	-1	-1	-1
Hijo derecho	1	4	-1	-1	-1	-1	-1	-1	-1	-1

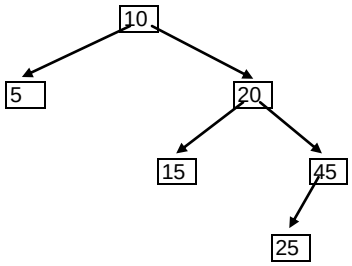

Figura 4.13. Árbol binario antes de eliminar.

Resultado:

	0	1	2	3	4	5	6	7	8	9
Clave	10	15	5	45	25	25	0	0	0	0
Hijo izquierdo	2	-1	-1	4	-1	-1	-1	-1	-1	-1
Hijo derecho	1	3	-1	-1	-1	-1	-1	-1	-1	-1

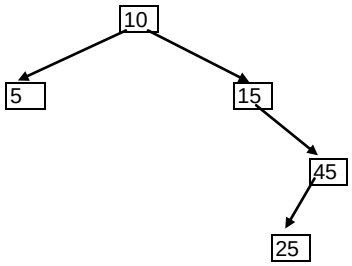

Figura 4.14. Árbol binario después de eliminar la clave 20.

Obsérvese que, al haber desplazado los elementos de la matriz hacia la izquierda, la clave 25 aparece repetida. De todas formas, dicho valor no se tendrá en cuenta, dado que la posición 5 sólo se alcanzará en caso de insertar una nueva clave (y se sustituirá dicha clave por su nuevo valor).

4.5. UTILIZACIÓN DE UN TAD ÁRBOL.

Los algoritmos explicados en las secciones anteriores implican el conocimiento interno de la estructura del árbol binario. Otro tanto podría haberse hecho en caso de que el árbol se hubiera implementado mediante cualquier otra alternativa posible.

Otra opción consiste en que el programador dispusiera de un Tipo Abstracto de Datos del que, dada su característica de **ocultamiento**, se desconoce absolutamente su implementación y, en consecuencia, solo es posible disponer de sus funcionalidades mediante un conjunto de especificaciones. A continuación se muestra un ejemplo de un TAD árbol binario de números enteros positivos:

```
package tadArbol;
public interface Arbol {
   void crearNodo ();
     /*Crea un nuevo nodo en el árbol, que queda apuntando a ese nodo*/
   int obtenerClave ();
     /*Devuelve la clave contenida en la raiz del árbol*/
   NodoArbol devolverRaiz ();
     /*Devuelve una referencia a la raiz del arbol*/
   NodoArbol devolverHijoIzquierdo ();
     /*Devuelve el hijo izquierdo del árbol*/
   NodoArbol devolverHijoDerecho ();
     /*Devuelve el hijo derecho del árbol*/
   void ponerClave (int Clave);
     /*pone la clave pasada como argumento en la raiz del árbol*/
   void ponerReferencia (NodoArbol nuevoArbol);
     /*Hace que el árbol apunte al mismo sitio que nuevoArbol*/
   void ponerHijoIzquierdo (NodoArbol arbolAsignado);
     /*Hace que el hijo izquierdo del arbol apunte ahora a arbolAsignado*/
   void ponerHijoDerecho (NodoArbol arbolAsignado);
     /*Hace que el hijo derecho del arbol, apunte ahora a arbolAsignado*/
   void asignarNulo ();
     /*Hace que el arbol tome el valor null*/
   boolean esNulo ();
     /*Devuelve true si el arbol tiene valor null y false en caso contrario*/
   boolean iguales (NodoArbol otroArbol);
     /*Devuelve true si la raiz del árbol y la de otroArbol apuntan al mismo
     sitio, false en caso contrario*/
}
```

Dado el TAD árbol anterior, se propone desarrollar un método (sumaClaves) que obtenga la suma de las claves del árbol.

```
   static int sumaClaves (Arbol arbol) {
      int resul = 0;
      Arbol izq = new TadArbol (), der = new TadArbol ();
      if (!arbol.esNulo()) {
         izq.ponerReferencia (arbol.devolverHijoIzquierdo ());
         der.ponerReferencia (arbol.devolverHijoDerecho ());
         resul = arbol.obtenerClave () + sumaClaves (izq) + sumaClaves (der);
      }
      return resul;
   }
```

4.5.1. Implementación del TAD Árbol

```
package tadArbol;

public class TadArbol implements Arbol {
   NodoArbol raiz;
   public TadArbol () {
      raiz = null;
   }
    /*Crea un nuevo nodo en el árbol, que queda apuntando a ese nodo*/
   public void crearNodo () {
      raiz = new NodoArbol ();
   }
   /*Devuelve la clave contenida en la raiz del árbol */
   public int obtenerClave () {
      int resul = 0;
      if (raiz != null)
         resul = raiz.clave;
      return resul;
   }
     /*Devuelve el hijo izquierdo del árbol*/
   public NodoArbol devolverHijoIzquierdo () {
      NodoArbol resul = null;
      if (raiz!= null)
         resul = raiz.iz;
      return resul;
   }
     /*Devuelve el hijo derecho del árbol*/
   public NodoArbol devolverHijoDerecho () {
      NodoArbol resul = null;
      if (raiz != null)
         resul = raiz.de;
      return resul;
   }
     /*pone la clave pasada como argumento en la raiz del árbol*/
   public void ponerClave (int clave) {
      if (raiz != null)
         raiz.clave = clave;
   }
     /*Hace que el árbol apunte al mismo sitio que nuevoArbol*/
   public void ponerReferencia (NodoArbol nuevoArbol) {
      raiz = nuevoArbol;
   }
     /*Hace que el hijo izquierdo del arbol apunte ahora a arbolAsignado*/
   public void ponerHijoIzquierdo (NodoArbol arbolAsignado) {
      if (raiz != null)
         raiz.iz = arbolAsignado;
      else System.out.println ("Error, el árbol está vacio");
   }
     /*Hace que el hijo derecho del arbol, apunte ahora a arbolAsignado*/
   public void ponerHijoDerecho (NodoArbol arbolAsignado) {
      if (raiz != null)
         raiz.de = arbolAsignado;
      else System.out.println ("Error, el árbol está vacio");
   }
```

```
    /*Hace que el arbol tome el valor null*/
   public void asignarNulo () {
      raiz = null;
   }
    /*Devuelve true si la raiz del arbol tiene valor null y false en caso
    contrario*/
   public boolean esNulo () {
      return raiz == null;
   }
    /*Devuelve true si la raiz del árbol apunta al mismo sitio que otroArbol,
    false en caso contrario*/
   public boolean iguales (NodoArbol otroArbol) {
      return raiz == otroArbol;
   }
    /*Devuelve una referencia a la raiz del arbol*/
   public NodoArbol devolverRaiz() {
      return raiz;
   }
}
```